Data Sieving and Collective I/O in ROMIO[*]

Rajeev Thakur   *William Gropp*   *Ewing Lusk*
Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439, USA
{thakur, gropp, lusk}@mcs.anl.gov

Preprint ANL/MCS-P723-0898
August 1998

Abstract

The I/O access patterns of parallel programs often consist of accesses to a large number of small, noncontiguous pieces of data. If an application's I/O needs are met by making many small, distinct I/O requests, however, the I/O performance degrades drastically. To avoid this problem, MPI-IO allows users to access a noncontiguous data set with a single I/O function call. This feature provides MPI-IO implementations an opportunity to optimize data access.

We describe how our MPI-IO implementation, ROMIO, delivers high performance in the presence of noncontiguous requests. We explain in detail the two key optimizations ROMIO performs: data sieving for noncontiguous requests from one process and collective I/O for noncontiguous requests from multiple processes. We describe how one can implement these optimizations portably on multiple machines and file systems, control their memory requirements, and also achieve high performance. We demonstrate the performance and portability with performance results for three applications—an astrophysics-application template (DIST3D), the NAS BTIO benchmark, and an unstructured code (UNSTRUC)—on five different parallel machines: HP Exemplar, IBM SP, Intel Paragon, NEC SX-4, and SGI Origin2000.

[*]This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Computational and Technology Research, U.S. Department of Energy, under Contract W-31-109-Eng-38; and by the Scalable I/O Initiative, a multiagency project funded by the Defense Advanced Research Projects Agency (contract number DABT63-94-C-0049), the Department of Energy, the National Aeronautics and Space Administration, and the National Science Foundation.

1 Introduction

Numerous studies of the I/O characteristics of parallel applications have shown that many applications need to access a large number of small, noncontiguous pieces of data from a file [1, 3, 9, 11, 12, 16]. For good I/O performance, however, the size of an I/O request must be large (on the order of megabytes). The I/O performance suffers considerably, on the other hand, if applications access data by making many small I/O requests. Such is the case when applications perform I/O by using the Unix `read` and `write` functions, which can access only a single contiguous chunk of data at a time.

MPI-IO, the I/O part of the MPI-2 standard [8], is a new interface designed specifically for portable, high-performance parallel I/O. To avoid the above-mentioned problem of many distinct, small I/O requests, MPI-IO allows users to specify the entire noncontiguous access pattern and read or write all the data with a single I/O function call. MPI-IO also allows users to specify collectively the I/O requests of a group of processes, thereby providing the implementation with even greater access information and greater scope for optimization.

In this paper we describe how our MPI-IO implementation, ROMIO, delivers high performance in the presence of noncontiguous I/O requests. ROMIO is a portable MPI-IO implementation that works on many different machines and file systems. We explain in detail the two key optimizations ROMIO performs: data sieving for noncontiguous requests from one process and collective I/O for noncontiguous requests from multiple processes. We describe how one can implement these optimizations portably on multiple machines and file systems, control their memory requirements, and also achieve high performance. We demonstrate the performance and portability with performance results for three applications on five different parallel machines: HP Exemplar, IBM SP, Intel Paragon, NEC SX-4, and SGI Origin2000. The three applications we used are the following:

1. DIST3D, a template representing the I/O access pattern in an astrophysics application, ASTRO3D [16], from the University of Chicago;

2. the NAS BTIO benchmark [5]; and

3. an unstructured code (which we call UNSTRUC) written by Larry Schoof and Wilbur Johnson of Sandia National Laboratories.

We note that ROMIO can perform the optimizations described in this paper only if users provide complete access information in a single function call. In [17], we explained how users can do so by using MPI's derived datatypes to create file views and by using the collective-I/O functions whenever possible. In this paper, we describe the optimizations in detail and provide extensive performance results.

The rest of this paper is organized as follows. Section 2 gives a brief overview of ROMIO. Data sieving is described in Section 3 and collective I/O in Section 4. Performance results are presented in Section 5, followed by conclusions in Section 6.

2 Overview of ROMIO

ROMIO is a freely available, high-performance, portable implementation of MPI-IO. The current version of ROMIO, 1.0.1, runs on the following machines: IBM SP; Intel Paragon; HP Exemplar; SGI Origin2000; NEC SX-4; other symmetric multiprocessors from HP, SGI, Sun, DEC, and IBM; and networks of workstations (Sun, SGI, HP, IBM, DEC, Linux, and FreeBSD). Supported file systems are IBM PIOFS, Intel PFS, HP HFS, SGI XFS, NEC SFS, NFS, and any Unix file system (UFS). ROMIO 1.0.1 includes everything defined in the MPI-2 I/O chapter except shared-file-pointer functions, split-collective-I/O functions, support for file interoperability, I/O error handling, and I/O error classes. ROMIO has been designed to be used with *any* MPI-1 implementation—both portable and vendor-specific implementations. It is currently included as part of two MPI implementations: MPICH and HP MPI.

A key component of ROMIO that enables such a portable MPI-IO implementation is an internal layer called ADIO [15]. ADIO, an abstract-device interface for I/O, is a mechanism for implementing multiple parallel-I/O APIs (application programming interfaces) portably on multiple file systems. We developed ADIO before MPI-IO became a standard, as a means to implement and experiment with various parallel-I/O APIs that existed at the time. We thought that such experimentation would help in the definition of a standard API.

ADIO consists of a small set of basic functions for *parallel* I/O. Any parallel-I/O API can be implemented portably on top of ADIO, and ADIO itself must be implemented separately on each different file system. ADIO thus separates the machine-dependent and machine-independent aspects involved in implementing an API. We used ADIO to implement Intel's PFS API and subsets of IBM's PIOFS and the original MPI-IO proposal [18] on PFS, PIOFS, Unix, and NFS file systems. By following such an approach, we achieved portability with very low overhead [15]. Now that MPI-IO has emerged as the standard, we use ADIO as a mechanism for implementing MPI-IO, as illustrated in Figure 1.

A similar abstract-device interface is used in MPICH [6] for implementing MPI portably.

3 Data Sieving

To reduce the effect of high I/O latency, it is critical to make as few requests to the file system as possible. Data sieving [14] is a technique that enables an implementation to make a few large, contiguous requests to the file system even if the user's request consists of several small, noncontiguous accesses.

Figure 2 illustrates the basic idea of data sieving. Assume that the user has made a single read request for five noncontiguous pieces of data. Instead of reading each noncontiguous piece separately, the implementation reads a single contiguous chunk of data starting from the first requested byte up to the last requested byte into a temporary buffer in memory. It then extracts the requested portions from the temporary buffer and places them in the user's buffer. The user's buffer happens to be contiguous in this example, but it could well be noncontiguous.

A potential problem with this simple algorithm is its memory requirement. The temporary buffer into which data is first read must be as large as the *extent* of the user's request,

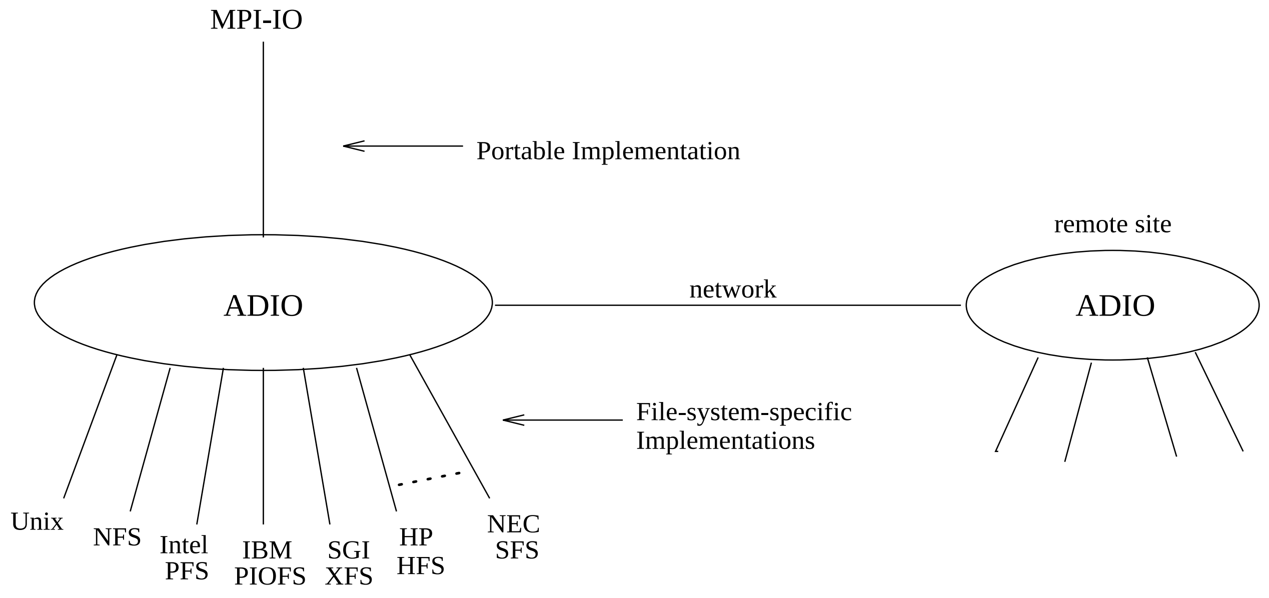

Figure 1: ROMIO Architecture: MPI-IO is implemented portably on top of an abstract-device interface called ADIO, and ADIO is optimized separately for different file systems.

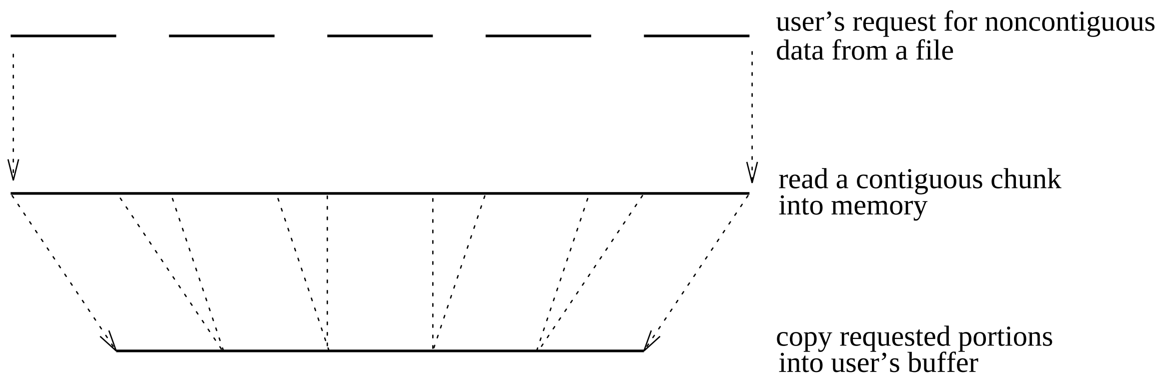
user's request for noncontiguous data from a file
read a contiguous chunk into memory
copy requested portions into user's buffer

Figure 2: Data sieving

where extent is defined as the total number of bytes between the first and last byte requested (including holes). The extent can potentially be very large—much larger than the amount of memory available for the temporary buffer—because the holes (unwanted data) between the requested data segments could be very large. The basic algorithm must therefore be modified to make its memory requirement independent of the extent of the user's request.

ROMIO uses a user-controllable parameter that defines the maximum amount of contiguous data that ROMIO can read at a time during data sieving. This value also represents the maximum size of the temporary buffer. By default, ROMIO sets the value of this parameter to 4 Mbytes, but the user can change it at run time via MPI-IO's hints mechanism. If the extent of the user's request is larger than the value of this parameter, ROMIO performs data sieving in parts, reading only as much data at a time as defined by the parameter.

The advantage of data sieving is that data is always accessed in large chunks, although at the cost of reading more data than needed. For many common access patterns, the holes between useful data are not unduly large, and the advantage of accessing large chunks far outweighs the cost of reading extra data. In some access patterns, however, the holes are so large that the cost of reading the extra data outweighs the cost of accessing large chunks. The BTIO benchmark (see Section 5), for example, has such an access pattern. An "intelligent" data-sieving algorithm can handle such cases as well. The algorithm can analyze the user's request and calculate the sizes of holes in it. Based on empirically determined prior knowledge of how large holes can get before data sieving is no longer beneficial, the algorithm can decide whether to perform data sieving or access each contiguous data segment separately. We plan to add this feature to ROMIO's data-sieving implementation.

Data sieving can similarly be used for writing data. A read-modify-write must, however, be performed to avoid destroying the data already present in the holes between contiguous data segments. For writing with data sieving, ROMIO first reads a contiguous chunk of data from the file into a temporary buffer in memory, copies data from the user's buffer into appropriate locations in the temporary buffer, and then writes the temporary buffer back to the file. The portion of the file being accessed must be locked during the read-modify-write to prevent concurrent updates by other processes.

ROMIO also uses another user-controllable parameter that defines the maximum amount of contiguous data that ROMIO can write at a time during data sieving. This parameter may have a smaller value than the one used for reading, because writing involves locking the region of the file being accessed. If the region being locked is too large, many processes remain idle waiting for the lock to be released. Consequently, parallelism in I/O is lost and performance decreases. On the other hand, if the region being locked is too small, there is greater parallelism, but the size of each I/O access also decreases, and performance is again adversely affected. In other words, there is a compromise between allowing greater concurrency and having large access sizes. We determined experimentally that a write size of 512 Kbytes provides a good trade-off between the two conflicting goals and gives good performance. ROMIO therefore sets the default value of the maximum buffer size for writing to 512 Kbytes. The user can, of course, change this value at run time.

4 Collective I/O

The preceding section explained how data sieving can be used to optimize I/O when the entire (noncontiguous) access information of a single process is known. Further optimization is possible if the implementation is given the entire access information of a group of processes. Such optimization is broadly referred to as collective I/O.

In many parallel applications, although each process may need to access several noncontiguous portions of a file, the requests of different processes are often interleaved and may together span large contiguous portions of the file. I/O performance can therefore be improved significantly by merging the requests of different processes and servicing the merged request, that is, by performing collective I/O.

Collective I/O can be performed in different ways and has been studied by many researchers in recent years. It can be done at the disk level (disk-directed I/O [7]), at the server level (server-directed I/O [10]), or at the client level (two-phase I/O [4]). Each method has its merits and demerits. Since ROMIO is a portable, user-level library with no separate I/O servers, it performs collective I/O at the client level. For this purpose, it uses a generalized version of the extended two-phase method described in [13].

4.1 Two-Phase I/O

Two-phase I/O was first proposed in [4] in the context of accessing distributed arrays from files. Consider the example of reading a two-dimensional array from a file into a (block,block) distribution in memory, as shown in Figure 3. Assume that the array is stored in the file in row-major order. As a result of the distribution in memory and the storage order in the file, the local array of each process is located noncontiguously in the file—each row of the local array of a process is separated by rows from the local arrays of other processes. If each process tries to read each row of its local array individually, the performance will be poor due to the large number of relatively small I/O requests. Note, however, that all processes together need to read the entire file, and two-phase I/O uses this fact to improve performance.

If the entire I/O access pattern of all processes is known to the implementation, the data can be accessed efficiently by splitting the access into two phases. In the first phase, processes access data assuming a distribution in memory that results in each process making a single, large, contiguous access. In this example, such a distribution is a row-block or (block,*) distribution. In the second phase, processes redistribute data among themselves to the desired distribution. The advantage of this method is that by making all file accesses large and contiguous, the I/O time is reduced significantly. The added cost of interprocess communication for redistribution is small compared with the savings in I/O time.

The basic two-phase method was extended in [13] to access sections of out-of-core arrays. Since MPI-IO is a general parallel-I/O interface, I/O requests in MPI-IO can represent *any* access pattern, not just arrays. The two-phase method in [4] must therefore be generalized to handle any noncontiguous I/O request. Such a generalized implementation of two-phase I/O, explained below, is used in ROMIO.

Two-phase I/O does increase the memory requirements of a program. For reading a

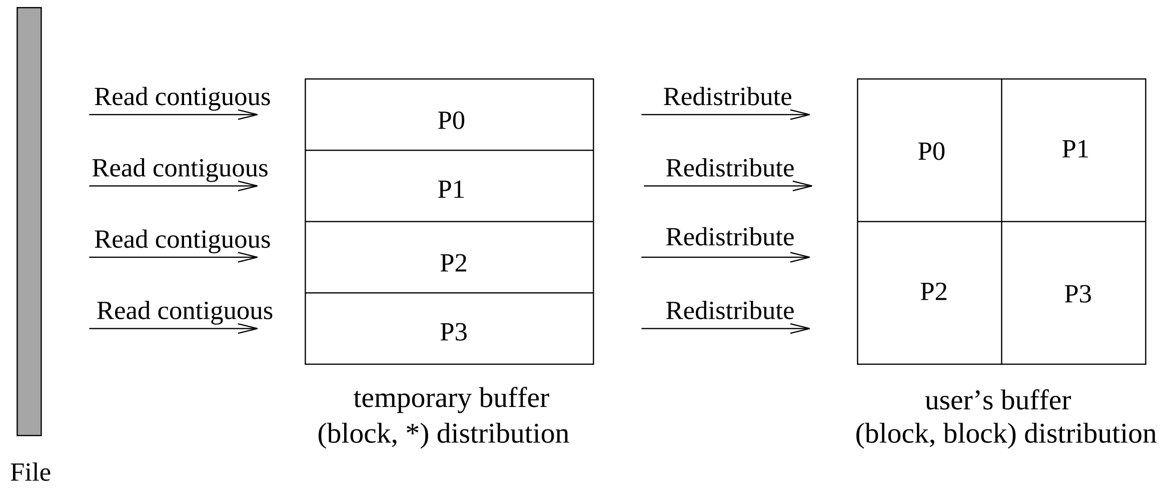

Figure 3: Reading a distributed array by using two-phase I/O

distributed array, for example, the amount of extra memory needed on each process (to store the data read in the first phase) is equal to the size of the local array itself. Since this amount of memory may not be available, the basic two-phase algorithm must be modified to read and communicate smaller parts of the array at a time. Similarly, on machines in which the I/O performance does not scale with the number of processes making simultaneous file accesses, it may be beneficial to have only a subset of processes perform I/O, with the remaining processes participating only in the redistribution phase. All these generalizations—any access pattern, fixed memory requirement, and variable number of processes performing I/O—are incorporated in ROMIO's collective-I/O implementation.

Figure 4 shows a simple example that illustrates how ROMIO performs a collective read. In this example, all processes perform I/O, and each process is assumed to have as much memory as needed for the temporary buffer.

4.2 Generalized Two-Phase I/O in ROMIO

ROMIO uses two user-controllable parameters for collective I/O: the number of processes that should directly access the file and the maximum size on each process of the temporary buffer needed for two-phase I/O. By default, ROMIO sets the number of processes that can directly access the file to the number of processes involved in the collective-I/O operation. The user can lower this number via MPI-IO's hints mechanism, but cannot increase it. The maximum buffer size is set to 4 Mbytes by default, and the user can change it to any number.

In MPI-IO, the collective-I/O function called by a process specifies the access information of that process only. If an MPI-IO implementation needs the access information of all processes participating in a collective-I/O operation, it must gather the information from those processes during the execution of the collective-I/O function. Note also that file accesses in this case refer to accesses from multiple processes to a *common* file.

We first explain the algorithm ROMIO uses for collective reads and then describe how the algorithm differs for collective writes.

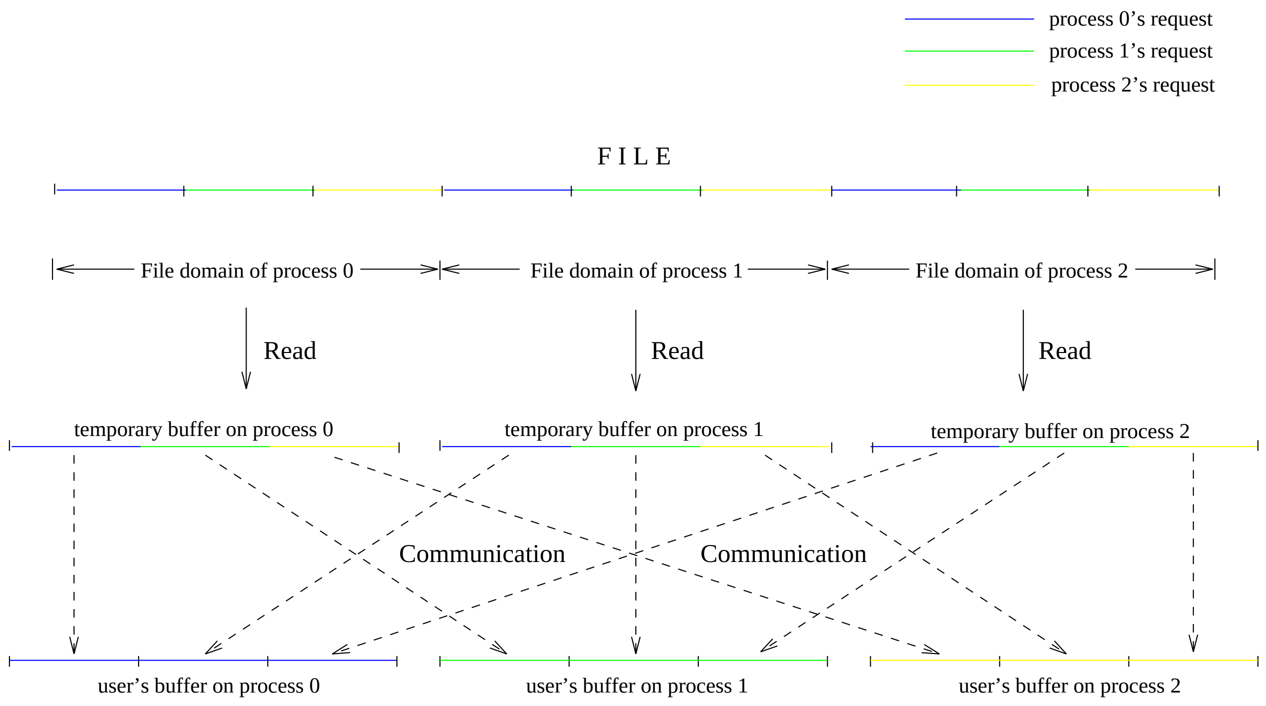

Figure 4: A simple example illustrating how ROMIO performs a collective read

4.2.1 Collective Reads

Each process first analyzes its own I/O request and creates a list of offsets and a list of lengths, where `length[i]` gives the number of bytes that the process needs from location `offset[i]` in the file. Each process also calculates the location of the first byte (start offset) and the location of the last byte (end offset) it needs from the file and then broadcasts these two offsets to other processes. As a result, each process has the start and end offsets of all processes.

In the next step, ROMIO tries to determine whether this particular access pattern can benefit from collective I/O. It checks whether the accesses of any processes are interleaved in the file; that is, for any two processes i and j, is the following expression true?

$$(\text{start-offset}_i < \text{end-offset}_j) \text{ AND } (\text{end-offset}_i > \text{end-offset}_j)$$

If the accesses of any two processes are not interleaved, ROMIO concludes that collective I/O will not improve performance for this particular access pattern, since the requests of processes cannot be merged. In such cases, ROMIO just calls the corresponding independent-I/O function on each process; the independent-I/O function uses data sieving to optimize noncontiguous requests.

If the accesses of different processes are interleaved in the file, ROMIO proceeds to perform collective I/O. It assigns portions of the file to each process such that in the I/O phase of the two-phase operation, a process will access data only from the portion of the file assigned to it. This portion of the file assigned to a process is called the process's *file domain.* If a process needs data located in another process's file domain, it will receive the data from the other process during the communication phase of the two-phase operation. Similarly, if this process's file domain contains data needed by other processes, it must send this data to those processes during the communication phase.

ROMIO assigns file domains as follows. It calculates the minimum of the start offsets and the maximum of the end offsets of all processes. The difference between these two offsets gives the total extent of the combined request of all processes. The file domain of each process is obtained by dividing this extent equally among the processes. For example, if the combined request of all processes spans from offset 100 to offset 399 in the file, and there are three processes, the file domain of process 0 will be from offset 100 to 199; the file domain of process 1 will be from offset 200 to 299; and the file domain of process 2 will be from offset 300 to 399.

When file domains are selected in this manner, the file domain of a process may not contain data needed by any process (e.g., if the access pattern has large holes). In such a case, the process will not perform any I/O and will participate only in communication. A more intelligent selection of file domains, based on analyzing the access pattern, can be used to ensure an even balance of the I/O workload among processes. We plan to implement such a scheme in ROMIO.

After the file domains are determined, each process calculates in which other process's file domain its own I/O request (or a portion of it) is located. For each such process, it creates a data structure containing a list of offsets and lengths that specify the data needed from the file domain of that process. It then sends this access information to the processes from which it expects to receive data. Similarly, other processes that need data from the file domain of this process send the corresponding access information to this process. After this exchange has taken place, each process knows what portions of its file domain are needed by other processes and by itself. It also knows which other processes are going to send the data that it needs.

The next step is to read and communicate the data. This step consumes the majority of the time, because all the I/O and data communication takes place here. Note that the communication in earlier steps involved only access information. The access information is usually much smaller than actual data, unless the access pattern is so irregular that an index is needed to represent the location of every basic datatype needed from the file.

As mentioned above, ROMIO performs the read-and-communicate step in several parts to reduce its memory requirement. Each process first calculates the offsets corresponding to the first and last bytes needed (by any process) from its file domain. It then divides the difference between these offsets by the maximum size allowed for the temporary buffer. The result is the number of times (`ntimes`) it needs to perform I/O. All processes then perform a global-maximum operation on `ntimes` to determine the maximum number of times (`max_ntimes`) any process needs to perform I/O. Even if a process has completed all the I/O needed from its own file domain, it may need to participate in communication operations thereafter to receive data from other processes. Each process must therefore be ready to participate in the communication phase `max_ntimes` number of times.

For each of the `ntimes` I/O operations, a process does the following operations: It checks if the current portion of its file domain (no larger than the maximum buffer size) has data that any process needs, including itself. If it does not have such data, the process does not need to perform I/O in this step; it then checks whether it needs to receive data from other processes, as explained below. If it does have such data, it reads with a single I/O function call all the data from the first offset to the last offset needed from this portion of the file domain into a temporary buffer in memory. The process effectively performs data sieving, as the data read may include some unwanted data. Now the process must send portions of the data read to processes that need them.

Since the one-sided communication defined in MPI-2 is not yet commonly supported by MPI implementations, we use the two-sided communication functions from MPI-1. Each process first informs other processes how much data it is going to send to each of them. The processes then exchange data by first posting all the receives as nonblocking operations, then posting all the nonblocking sends, and finally waiting for all the nonblocking communication to complete. MPI derived datatypes are used to send noncontiguous data directly from the temporary buffer to the destination process. On the receive side, if the user has asked for data to be placed contiguously in the user-supplied buffer, the data is received directly into the user's buffer. If data is to be placed noncontiguously, the process first receives data into a temporary buffer and then copies it into the user's buffer. (Since data is received in parts over multiple communication operations from different processes, we found this approach easier than creating derived datatypes on the receive side.)

Each process performs I/O and communication `ntimes` number of times and then participates only in the communication phase for the remaining `(max_ntimes - ntimes)` number of times. In some of these remaining communication steps, a process may not receive any data; nevertheless, it needs to *check* whether it will receive data in a particular step.

4.2.2 Collective Writes

The algorithm for collective writes is similar to the one for collective reads, except that the first phase of the two-phase operation is communication and the second phase is I/O. In the I/O phase, each process checks to see whether there are any holes in the data it currently needs to write. If there are holes, it performs a read-modify-write; otherwise it performs only a write. During the read-modify-write, a process need not lock the region of the file being accessed (unlike in independent I/O), because the process is assured that no other process involved in the collective-I/O operation will directly try to access the data located in this process's file domain. The process is also assured that concurrent writes from processes not involved in this collective-I/O operation will not occur, because MPI-IO's consistency semantics [8] do not automatically guarantee consistency for such writes. (In such cases, users must use `MPI_File_sync` and ensure that the operations are not concurrent.)

4.2.3 Performance Issues

Even if I/O is performed in large contiguous chunks, the performance of the collective-I/O implementation can be significantly affected by the amount of buffer copying and communication. We were able to improve ROMIO's collective-I/O performance by as much as 50% on some machines by tuning the implementation to

1. minimize buffer copying and

2. minimize the number of communication calls and use the right set of MPI communication primitives.

Initially, in each of the communication steps, we always received data into a temporary buffer and then copied it into the user's buffer. We realized later that this copy is needed only when the user's buffer is to be filled noncontiguously. In the contiguous case, data can be received directly into the appropriate location in the user's buffer. We similarly experimented with different ways of communicating data in MPI and measured the effect on overall collective-I/O performance with different MPI implementations and on different machines. We selected nonblocking communication with the receives posted first and then the sends, which performs the best on most systems. It may be possible, however, to tune the communication further on some machines by posting the sends before the receives or by using MPI's persistent requests.

We believe that MPI-2's one-sided communication can be used effectively for the communication involved in collective I/O, and we plan to use it when it is available in MPI implementations.

4.2.4 Portability Issues

We were able to implement these optimizations portably and without sacrificing performance by

1. using ADIO as a portability layer for I/O (see Section 2) and

2. using MPI for communication.

Data sieving and collective I/O are implemented as ADIO functions [15]: Data sieving is used in the ADIO functions that read/write noncontiguous data, and collective I/O is used in ADIO's collective-I/O functions. Both these optimizations ultimately make contiguous I/O requests to the underlying file system, which are implemented by using ADIO's contiguous-I/O functions. The contiguous-I/O functions, in turn, are implemented using the appropriate file-system call for each different file system. (For example, on file systems that support 64-bit file sizes, we use the functions needed for files of 64-bit size. On SGI's XFS file system and on the HP Exemplar's HFS file system, we use the recommended functions `pread64` and `pwrite64` that also take the file offset as an argument; that is, no seeks are needed.)

5 Performance Measurements

We first briefly describe the three applications used in the performance experiments, the machines on which we ran the applications, and the set of experiments performed. We then present the performance results.

5.1 Applications

The first application we used is DIST3D, a template representing the I/O access pattern in an astrophysics application, ASTRO3D [16], from the University of Chicago. It measures the performance of reading/writing a three-dimensional array distributed in a (block,block,block) fashion among processes from/to a file containing the global array in row-major order.

The second application is the BTIO benchmark [5] from NASA Ames Research Center, which simulates the I/O required by a time-stepping flow solver that periodically writes its solution matrix. The benchmark only performs writes, but we modified it to also perform reads, in order to measure the read bandwidths. In BTIO, a three-dimensional array (actually four-dimensional, but the first dimension has only five elements and is not distributed) is distributed among processes by using a multipartition distribution [2]. In this distribution, each process is responsible for several disjoint sub-blocks of points (cells) of the grid. The cells are arranged such that, for each direction of the solve phase, the cells belonging to a certain process will be evenly distributed along the direction of solution. (Note that this distribution is different from the (block,block,block) distribution of the first application.) The array elements are stored in the file in an order corresponding to a column-major ordering of the global array.

The third application we used is an unstructured code (which we call UNSTRUC) written by Larry Schoof and Wilbur Johnson of Sandia National Laboratories. It is a synthetic benchmark that emulates the I/O access pattern in unstructured-grid applications. It generates a random irregular mapping from the local one-dimensional array of a process to a global array in a common file shared by all processes. The mapping specifies where each element of the local array is located in the global array. The size of each element can also be varied in the program.

5.2 Machines

We ran the code portably and measured the performance on five different parallel machines: the HP Exemplar and SGI Origin2000 at the National Center for Supercomputing Applications (NCSA), the IBM SP at Argonne National Laboratory, the Intel Paragon at California Institute of Technology, and the NEC SX-4 at the National Aerospace Laboratory (NLR) in Holland. These machines cover almost the entire spectrum of state-of-the-art high-performance systems, and they represent distributed-memory, shared-memory, and parallel vector architectures. They also represent a wide variation in I/O architecture, from the "traditional" parallel file systems on distributed-memory machines like the SP and Paragon, to the so-called high-performance file systems on shared-memory machines like the Origin2000, Exemplar, and SX-4.

We used the native file systems on each machine: HFS on the Exemplar, XFS on the Origin2000, PIOFS on the SP, PFS on the Paragon, and SFS on the SX-4. At the time we performed the experiments, these file systems were configured as follows: HFS on the Exemplar was configured on twelve disks; XFS on the Origin2000 had two RAID units with SCSI-2 interfaces; the SP had four servers for PIOFS, and each server had four SSA disks attached to it in one SSA loop; the Paragon had 64 I/O nodes for PFS, each with an individual Seagate disk; and SFS on the NEC SX-4 was configured on a single RAID unit comprising sixteen SCSI-2 data disks.

5.3 Experiments

We measured the I/O performance of these applications by using MPI-IO functions to perform I/O in three different ways as follows:

Unix-style accesses Separate MPI-IO function calls to access each individual contiguous piece of data.

Data sieving Create a file view to describe a noncontiguous access pattern and use a single *independent* MPI-IO function to access data.

Collective I/O Create a file view to describe a noncontiguous access pattern, and use a single *collective* MPI-IO function to access data.

The space of experimentation is very large, with many parameters that can be varied. While it would be interesting to see the effect of scaling the problem size and the number of processors, we chose to limit the experiments to large problem sizes on a large number of processors in order to limit the number of results to be gathered and presented. These experiments also give us an idea of the maximum I/O bandwidth that can realistically be achieved on these machines—a fact we were interested in knowing.

We measured the write performance without explicitly calling an `MPI_File_sync` to flush all cached data to disk. Some of the performance results may therefore include caching performed by the file system. We did not include `MPI_File_sync` in the measurements because users most often do not perform a file sync; they just open, read/write, and close.

In all experiments, we used the default values of the sizes of the buffers ROMIO uses for data sieving and collective I/O (see Sections 3 and 4). We also used the default values of the file-striping parameters on all file systems. On PFS and PIOFS the default striping unit was 64 Kbytes. We note that ROMIO does not use the logical-views feature of PIOFS; that is, a PIOFS file is considered as a linear sequence of bytes.

On each machine, we used as many processors as we could reasonably access. We also tried to use the same number of processors on a given machine for each application but were at times constrained by the application's requirements: BTIO requires that the number of processors be a perfect square, whereas UNSTRUC requires that the number of processors be a power of two. On some machines, therefore, we could not use the same number of processors for both BTIO and UNSTRUC; for example, on the NEC SX-4 we had to run BTIO on 9 processors and UNSTRUC on 8 processors. For UNSTRUC, we used a smaller grid size on the Origin2000 because of memory limitations.

5.4 Results

Tables 1 and 2 show the read and write bandwidths for DIST3D. The performance of Unix-style accesses was, in general, very poor. By using data sieving instead, we observed an improvement ranging from 162% (HP Exemplar) to 45,252% (NEC SX-4) for reads. The improvement in write performance ranged from 0% (IBM SP) to 12,045% (NEC SX-4). No performance improvement was observed on the SP for writing, because ROMIO cannot perform data sieving for writing on the SP's PIOFS file system, as PIOFS does not support

Table 1: Read performance of DIST3D (array size 512 × 512 × 512 integers, file size 512 Mbytes)

		Bandwidth (Mbytes/s)		
Machine	Processors	Unix-style	Data Sieving	Collective I/O
HP Exemplar	64	5.42	14.2	68.2
IBM SP	64	2.13	11.9	90.2
Intel Paragon	256	3.01	9.50	132
NEC SX-4	8	0.71	322	563
SGI Origin2000	32	14.0	118	175

Table 2: Write performance of DIST3D (array size 512 × 512 × 512 integers, file size 512 Mbytes)

		Bandwidth (Mbytes/s)		
Machine	Processors	Unix-style	Data Sieving	Collective I/O
HP Exemplar	64	0.54	1.25	50.7
IBM SP	64	1.85	1.85	57.6
Intel Paragon	256	1.12	3.33	183
NEC SX-4	8	0.62	75.3	447
SGI Origin2000	32	5.06	13.1	66.7

file locking. On PIOFS, ROMIO therefore translates noncontiguous, independent, write requests into multiple Unix-style accesses.

The performance improvement with collective I/O was much more significant. The improvement in read performance was as high as 79,196% over Unix-style accesses (NEC SX-4) and as high as 1,289% over data sieving (Intel Paragon). The improvement in write performance was as high as 71,997% over Unix-style accesses (NEC SX-4) and as high as 3,956% over data sieving (HP Exemplar).

The BTIO benchmark comes in three problem sizes: Class A (64^3), Class B (102^3), and Class C (162^3). We present results for the largest case, Class C, in Tables 3 and 4. For BTIO, Unix-style accesses perform better than data sieving on three out of the five machines. The reason is that the access pattern of BTIO is such that the holes between data segments needed by a process are large—more than five times the size of the data segment. As a result, a lot of unwanted data is accessed during data sieving, resulting in lower performance than with Unix-style accesses. As mentioned in Section 3, an intelligent data-sieving algorithm can detect such large holes and internally perform Unix-style accesses. ROMIO's data sieving algorithm does not currently do this, however.

Collective I/O performed extremely well on BTIO, as no unwanted data is accessed in collective I/O, and all accesses are large. The performance improvement was as high as

Table 3: Read performance of BTIO (Class C, problem size $5 \times 162 \times 162 \times 162$ double precision $\approx$ 162 Mbytes)

		Bandwidth (Mbytes/s)		
Machine	Processors	Unix-style	Data Sieving	Collective I/O
HP Exemplar	64	6.35	5.84	44.2
IBM SP	64	2.73	1.66	80.6
Intel Paragon	256	2.28	1.23	82.0
NEC SX-4	9	1.26	116	645
SGI Origin2000	36	12.1	37.0	107

Table 4: Write performance of BTIO (Class C, problem size $5 \times 162 \times 162 \times 162$ double precision $\approx$ 162 Mbytes)

		Bandwidth (Mbytes/s)		
Machine	Processors	Unix-style	Data Sieving	Collective I/O
HP Exemplar	64	0.86	0.50	29.7
IBM SP	64	2.21	2.21	38.6
Intel Paragon	256	1.37	0.45	98.8
NEC SX-4	9	0.99	29.9	591
SGI Origin2000	36	7.93	2.90	67.2

51,090% over Unix-style accesses for reading and 59,597% for writing, both on the NEC SX-4.

Tables 5 and 6 show the read and write bandwidths for UNSTRUC. In this application, the I/O access pattern is irregular, and the granularity of each access is very small (64 bytes). Unix-style accesses are not feasible for this kind of application, as they take an excessive amount of time. We therefore do not present results for Unix-style accesses for UNSTRUC. Since data sieving cannot be performed for writing on PIOFS, we also do not present results for level-2 writes on PIOFS. Collective I/O again performed much better than independent I/O with data sieving, the only exception being for reads on the NEC SX-4. In this case, because of the high read bandwidth of NEC's Supercomputing File System (SFS), data sieving by itself outperformed the extra communication required for collective I/O.

6 Conclusions

For parallel applications to achieve high I/O performance, it is critical that the parallel-I/O system be able to deliver high performance even for noncontiguous access patterns. We have described two optimizations that enable an MPI-IO implementation to deliver high performance even if the user's request consists of many small, noncontiguous accesses.

Table 5: Read performance of UNSTRUC

			Bandwidth (Mbytes/s)	
Machine	Processors	Grid Points	Data Sieving	Collective I/O
HP Exemplar	64	8 million	3.15	35.0
IBM SP	64	8 million	1.63	73.3
Intel Paragon	256	8 million	1.18	78.4
NEC SX-4	8	8 million	152	101
SGI Origin2000	32	4 million	30.0	80.8

Table 6: Write performance of UNSTRUC

			Bandwidth (Mbytes/s)	
Machine	Processors	Grid Points	Data Sieving	Collective I/O
HP Exemplar	64	8 million	0.18	22.1
IBM SP	64	8 million	xx	37.8
Intel Paragon	256	8 million	0.22	94.9
NEC SX-4	8	8 million	16.8	81.5
SGI Origin2000	32	4 million	1.33	59.2

For the applications we considered, collective I/O performed significantly better than both data sieving and Unix-style accesses. Data sieving performed much better than Unix-style accesses for DIST3D and UNSTRUC. For BTIO, on some machines, Unix-style accesses performed better than data sieving, because of large holes between data segments accessed by each process in BTIO.

The implementation of data sieving and collective I/O must be carefully tuned to minimize the overhead of buffer copying and interprocess communication. Otherwise, these overheads can impact performance significantly.

To carry out these optimizations, an MPI-IO implementation needs some amount of temporary buffer space, which reduces the total amount of memory available to the application. The optimizations can, however, be performed with a constant amount of buffer space that does not increase with the size of the user's request. Our results demonstrate that by allowing the MPI-IO implementation to use as little as 4 Mbytes of buffer space, which is a small amount on today's high-performance machines, users can gain orders of magnitude improvement in I/O performance.

We note that MPI-IO also defines split collective-I/O functions, which are a form of nonblocking collective-I/O functions. The user can call a "begin" function to start the collective-I/O operation and an "end" function to complete the operation. The implementation is free to implement the collective-I/O operation either entirely during the begin function or entirely during the end function or in the "background," between the begin and end functions. We are currently investigating how to perform collective I/O in the background by using threads, so as to successfully overlap collective I/O with other computation and communication going on in the user's program.

Acknowledgments

We thank Larry Schoof and Wilbur Johnson for providing the unstructured code used in this paper.

References

[1] S. Baylor and C. Wu. Parallel I/O Workload Characteristics Using Vesta. In R. Jain, J. Werth, and J. Browne, editors, *Input/Output in Parallel and Distributed Computer Systems*, chapter 7, pages 167–185. Kluwer Academic Publishers, 1996.

[2] J. Bruno and P. Cappello. Implementing the Beam and Warming Method on the Hypercube. In *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications*, January 1988.

[3] P. Crandall, R. Aydt, A. Chien, and D. Reed. Input-Output Characteristics of Scalable Parallel Applications. In *Proceedings of Supercomputing '95*. ACM Press, December 1995.

[4] J. del Rosario, R. Bordawekar, and A. Choudhary. Improved Parallel I/O via a Two-Phase Run-time Access Strategy. In *Proceedings of the Workshop on I/O in Parallel Computer Systems at IPPS '93*, pages 56–70, April 1993. Also published in *Computer Architecture News*, 21(5):31–38, December 1993.

[5] S. Fineberg, P. Wong, B. Nitzberg, and C. Kuszmaul. PMPIO—A Portable Implementation of MPI-IO. In *Proceedings of the Sixth Symposium on the Frontiers of Massively Parallel Computation*, pages 188–195. IEEE Computer Society Press, October 1996.

[6] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A High-Performance, Portable Implementation of the MPI Message-Passing Interface Standard. *Parallel Computing*, 22(6):789–828, September 1996.

[7] D. Kotz. Disk-directed I/O for MIMD Multiprocessors. *ACM Transactions on Computer Systems*, 15(1):41–74, February 1997.

[8] Message Passing Interface Forum. *MPI-2: Extensions to the Message-Passing Interface.* July 1997. On the World-Wide Web at `http://www.mpi-forum.org/docs/docs.html`.

[9] N. Nieuwejaar, D. Kotz, A. Purakayastha, C. Ellis, and M. Best. File-Access Characteristics of Parallel Scientific Workloads. *IEEE Transactions on Parallel and Distributed Systems*, 7(10):1075–1089, October 1996.

[10] K. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett. Server-Directed Collective I/O in Panda. In *Proceedings of Supercomputing '95*. ACM Press, December 1995.

[11] E. Smirni, R. Aydt, A. Chien, and D. Reed. I/O Requirements of Scientific Applications: An Evolutionary View. In *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing*, pages 49–59. IEEE Computer Society Press, 1996.

[12] E. Smirni and D. Reed. Lessons from Characterizing the Input/Output Behavior of Parallel Scientific Applications. *Performance Evaluation: An International Journal*, 33(1):27–44, June 1998.

[13] R. Thakur and A. Choudhary. An Extended Two-Phase Method for Accessing Sections of Out-of-Core Arrays. *Scientific Programming*, 5(4):301–317, Winter 1996.

[14] R. Thakur, A. Choudhary, R. Bordawekar, S. More, and S. Kuditipudi. Passion: Optimized I/O for Parallel Applications. *Computer*, 29(6):70–78, June 1996.

[15] R. Thakur, W. Gropp, and E. Lusk. An Abstract-Device Interface for Implementing Portable Parallel-I/O Interfaces. In *Proceedings of the 6th Symposium on the Frontiers of Massively Parallel Computation*, pages 180–187, October 1996.

[16] R. Thakur, W. Gropp, and E. Lusk. An Experimental Evaluation of the Parallel I/O Systems of the IBM SP and Intel Paragon Using a Production Application. In *Proceedings of the 3rd International Conference of the Austrian Center for Parallel Computation (ACPC) with Special Emphasis on Parallel Databases and Parallel I/O*, pages 24–35. Lecture Notes in Computer Science 1127. Springer-Verlag., September 1996.

[17] R. Thakur, W. Gropp, and E. Lusk. A Case for Using MPI's Derived Datatypes to Improve I/O Performance. In *Proceedings of SC98: High Performance Networking and Computing*, November 1998. To appear.

[18] The MPI-IO Committee. MPI-IO: A Parallel File I/O Interface for MPI, Version 0.5. World-Wide Web `http://lovelace.nas.nasa.gov/MPI-IO`, April 1996.